Parallel Model-Based Diagnosis

Kostyantyn Shchekotykhin, Dietmar Jannach, and Thomas Schmitz

Abstract Model-Based Diagnosis (MBD) is a general-purpose computational approach to determine why a system under observation, e.g., an electronic circuit or a software program, does not behave as expected. MBD approaches utilize knowledge about the system's expected behavior if all of its components work correctly. In case of an unexpected behavior they systematically explore the possible reasons, i.e., diagnoses, for the misbehavior. Such diagnoses are determined through systematic or heuristic search procedures which often use MBD-specific rules to prune the search space. In this chapter we review approaches that rely on parallel or distributed computations to speed up the diagnostic reasoning process. Specifically, we focus on recent parallelization strategies that exploit the capabilities of modern multi-core computer architectures and report results from experimental evaluations to shed light on the speedups that can be achieved by parallelization for various MBD applications.

1 Introduction

1.1 Background

Model-Based Diagnosis (MBD) is a subfield of Artificial Intelligence that focuses on automated reasoning methods that are capable of generating hypotheses and

Kostyantyn Shchekotykhin
Institute for Applied Informatics, Alpen-Adria-Universität Klagenfurt, Austria,
e-mail: konstantin.schekotihin@aau.at

Dietmar Jannach
Department of Computer Science, TU Dortmund, Germany,
e-mail: dietmar.jannach@tu-dortmund.de

Thomas Schmitz
Department of Computer Science, TU Dortmund, Germany,
e-mail: thomas.schmitz@tu-dortmund.de

explanations why a system under observation is not behaving as expected. The term "model-based" means that the diagnostic inference process is based on knowledge (i.e., a model) of how the system and its components work, which makes it possible to simulate the system's behavior in order to test alternative hypotheses.

MBD techniques were pioneered in the 1980s and many of the proposals at that time were centered on the application domain of digital circuits [1, 2, 3]. The model in such cases consists of knowledge about (a) the normal and expected behavior of the components of the circuit (e.g., how an AND-gate works when it functions correctly), and (b) how the components are interconnected. This knowledge can then be used to simulate the behavior of the circuit and to compute the *expected outputs* for a given set of inputs. The resulting simulated behavior is then contrasted with the real and *observed behavior* of the system. Whenever there is a discrepancy between the expected and the observed outputs, the task of an MBD system is to determine which parts of the analyzed system *can be* responsible for the observed outputs.

The predominant algorithmic approach to find one or more sets of components that can be responsible for the observed faulty outputs is to systematically test different hypotheses about the (binary) health state of each of the components. One main and computationally complex part of the diagnostic reasoning process is therefore a search process in which the search space in principle consists of all possible subsets of the components of the analyzed system. In much research work the search process itself is guided by so-called "conflicts," which are typically comparatively small subsets of the system's components, which – according to the simulation – cannot all be working correctly, i.e., at least one of them must be faulty. These conflicts can help to significantly reduce the search space. Their computation can however also be computationally demanding. But even if all conflicts were known in advance, finding all possible diagnoses using the known conflicts corresponds to finding a solution to a set cover (hitting set) problem, leading to an NP-hard search problem.

In this chapter we review existing work that approaches the problem of the computational complexity of model-based reasoning processes by parallelizing parts of the reasoning and search processes.

1.2 Outline of the Chapter

The chapter is organized as follows. Next, in Section 2, we will review the formal and logic-based characterization of the Model-Based Diagnosis problem and a conflict-directed, sound and complete tree-based search method as introduced by Reiter in [3]. Reiter's domain-independent problem formalization is the basis for most of the works discussed in the chapter. Its generality is also one of the reasons for the success of MBD techniques and why they are still relevant today. MBD techniques are in fact not limited to electronic circuits, but have been applied over the last three decades in particular to a variety of software artifacts including logic programs, ontologies, process specifications, special purpose and general-purpose languages such as VHDL or Java, and recently also to spreadsheet programs [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14].

Section 3 then discusses alternative ways of finding diagnoses more quickly than with Reiter's approach. Typically, these methods make some sort of assumptions to make the problem easier to solve, e.g., by requiring that the diagnosed system has a certain structure or by focusing only on certain subsets of all existing diagnoses, thereby sacrificing the completeness of the algorithm. In this section we will also discuss so called "direct" methods that encode the diagnosis problem in a form that can be directly processed by specific inference engines such as a SAT solver without explicitly creating a diagnosis search tree.

Since diagnostic reasoning is predominantly a tree search problem, we will review general strategies for search parallelization (e.g., tree decomposition or window-based processing) and analyze the applicability of parallelized versions of general search strategies such as A^* to the Model-Based Diagnosis problem in Section 4.

Section 5 then presents recent techniques for parallelizing the MBD reasoning process on multi-core machines. The main focus will be on sound and complete diagnosis approaches and on methods that parallelize Reiter's tree search method in different ways. Selected results of empirical evaluations that were made using a number of benchmark problems are then presented in Section 6. These results help us to quantify the possible gains that can be obtained by running the search process on parallel threads on one machine.

Finally, in Section 7, we will discuss a number of alternative parallelization approaches that are not based on Reiter's HS-tree algorithm.

2 Reiter's Diagnosis Framework

In this section, we will summarize the formal characterization of the MBD problem as proposed by Reiter in [3]. The goal of Reiter's work was to provide a generic formalization that allows one to diagnose any system whose behavior can be modeled by a set of first-order sentences. Given a formal model describing the "normal" behavior of the system under observation, the general task of an MBD algorithm is to analyze the possible fault reasons, whenever the system does not behave as expected. The starting points for this analysis are therefore discrepancies between the system's outputs as predicted by the model and the observed outputs.

2.1 Example: A Diagnosis Problem Instance

Let us consider the binary half adder shown in Figure 1 as an example. This simple digital circuit whose behavior we know when it works normally, has two inputs A and B, two outputs S and C, an AND-gate A_1, and an XOR-gate X_1.

The half-adder can be described by first-order sentences as follows. First, we describe the expected behavior of the two types of logic gates, where *in(x,1)* denotes the first input of component x, *in(x,2)* its second input, and *out(x)* refers to the output

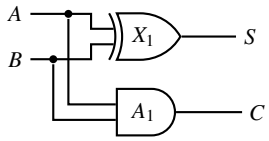

Fig. 1: Half-adder with inputs A and B, and outputs S and C

of the component. The *and* and *xor* functions implement boolean conjunction and exclusive disjunction respectively.

$$\forall x : \text{AND}(x) \rightarrow [out(x) = and(in(x,1), in(x,2))]$$
$$\forall x : \text{XOR}(x) \rightarrow [out(x) = xor(in(x,1), in(x,2))]$$

The components A_1 and X_1 and their wiring are modeled next.

$$\text{AND}(A_1), \quad \text{XOR}(X_1)$$
$$in(A_1,1) = in(X_1,1), \quad in(A_1,2) = in(X_1,2)$$

Finally, we state that the inputs of X_1 can only be 0 or 1 (which then also applies to A_1 according to the wiring of our circuit).

$$in(X_1,1) = 0 \vee in(X_1,1) = 1, \quad in(X_1,2) = 0 \vee in(X_1,2) = 1$$

Now let us assume that the following inputs to the system were provided, which are again described as a set of first-order sentences: $in(X_1,1) = 1,\ in(X_1,2) = 0$. The observed outputs were however $out(X_1) = 0,\ out(A_1) = 0$. This is obviously unexpected, since the output of the XOR-gate X_1 should be 1.

Having completed our description of the system and the inputs and outputs, we can feed all the first-order sentences modeled so far into a *theorem prover* (TP), which will find that the model is inconsistent. Consequently, an abnormal behavior has been detected.

However, the presented way of modeling does not easily allow us to find the true cause of the problem, i.e., that gate X_1 is broken. In order to find the possible causes of a problem, typical MBD systems test different assumptions about the faultiness of individual components. To be able to systematically explore the possible causes, Reiter proposes a modeling approach that uses a unary predicate $\text{AB}(\cdot)$ to denote that a component is "abnormal". Following this approach, we reformulate the first set of sentences of our model as follows.

$$\forall x : \neg\text{AB}(x) \rightarrow (\text{AND}(x) \rightarrow [out(x) = and(in(x,1), in(x,2))])$$
$$\forall x : \neg\text{AB}(x) \rightarrow (\text{XOR}(x) \rightarrow [out(x) = xor(in(x,1), in(x,2))])$$

The resulting model – in combination with the observations (i.e., the inputs and outputs) – allows us to test different assumptions about the correctness of the

components. For instance, if we assume that all components are working properly $\{\neg\text{AB}(X_1), \neg\text{AB}(A_1)\}$, the theorem prover will find that the model is inconsistent given the observations. However, under the assumption $\{\text{AB}(X_1), \neg\text{AB}(A_1)\}$, i.e., X_1 is faulty, we will find that the model is consistent with the observations. Assuming that X_1 is not working correctly therefore explains the observed outputs and the set $\{X_1\}$, which is a subset of the system's components, would thus be what is called a diagnosis in Reiter's framework.

Finally, the main advantage of the described modeling approach using "abnormal" predicates is that we can determine such diagnoses by systematically or heuristically varying our assumptions about the faultiness of the components.

2.2 Diagnoses and Conflicts

Formally, the principled approach to Model-Based Diagnosis by Reiter can be summarized as follows.

Definition 1. (Diagnosis problem instance) Let $(\text{SD}, \text{COMPS}, \text{OBS})$ be a triple, where SD and OBS are finite sets of first-order sentences which encode a system description and observations, respectively, and COMPS is a finite set of constants that represent the system's components.

$(\text{SD}, \text{COMPS}, \text{OBS})$ is a *diagnosis problem instance* (DPI) iff (1) SD is consistent, (2) OBS is consistent, and (3) $\text{SD} \cup \text{OBS} \cup \{\neg\text{AB}(c) \mid c \in \text{COMPS}\}$ is inconsistent.

Definition 2. (Diagnosis problem) Given a DPI $(\text{SD}, \text{COMPS}, \text{OBS})$, the diagnosis problem is to find a subset-minimal set $\Delta \subseteq \text{COMPS}$, called a *diagnosis*, such that $\text{SD} \cup \text{OBS} \cup \{\text{AB}(c) \mid c \in \Delta\} \cup \{\neg\text{AB}(c) \mid c \in \text{COMPS} - \Delta\}$ is consistent.

According to Definition 2, we are only interested in *minimal* diagnoses, i.e., diagnoses which contain no superfluous elements and which are thus not supersets of other diagnoses. Whenever we use the term *diagnosis* in the remainder of the chapter, we mean *minimal diagnosis*. Whenever we want to refer to non-minimal diagnoses, we will explicitly mention this fact.

Finding a diagnosis can in theory be done by simply trying out all possible subsets of COMPS and checking their consistency with the observations. A simple tree search algorithm that enumerates the possible assumptions in a breadth-first manner can be used for that purpose. Consider Algorithm 1, which takes a problem instance DPI as an input and returns one single diagnosis. The algorithm starts with the simple assumption that all components work properly. In every iteration it uses Definition 2 to test whether the current assumption Δ is a diagnosis. If this is not the case, the algorithm extends the search frontier by all supersets of Δ by adding one component from the set $\text{COMPS} - \Delta$. The breadth-first order guarantees the subset-minimality of the returned set of faulty components Δ.

Algorithm 1 is obviously very inefficient because it exhaustively enumerates all possible assumptions. In many real-world scenarios such an algorithm would make

Algorithm 1: Tree search algorithm

```
   Input: DPI (SD, COMPS, OBS)
   Output: A diagnosis Δ
1  closed ← ∅ ;                                  /* Maintain list of work already done */
2  frontier ← {∅} ;                              /* Initialize the search frontier */
3  while frontier ≠ ∅ do
       /* Get and remove the first element of the frontier                */
4      Δ ← Pop(frontier);
5      if Δ ∉ closed then                                          /* Skip the node */
6          closed ← closed ∪ {Δ};
7          if SD ∪ OBS ∪ {AB(c) | c ∈ Δ} ∪ {¬AB(c) | c ∈ COMPS − Δ} is consistent then
8              return Δ ;                                  /* A diagnosis is found */
           /* Generate successors of Δ and add them to the frontier          */
9          frontier ← frontier ∪ {Δ ∪ {c} | c ∈ (COMPS − Δ)}
10 return failure;                                    /* Provided triple is not a DPI */
```

many assumptions about the faultiness of components that are in fact irrelevant. This can for instance be the case if the unexpected behavior is observed only for a certain part of a physical system. Therefore, Reiter [3] proposes a more efficient procedure based on the concept of *conflicts*. The main idea is to focus only on those components that are actually involved in an inconsistency and to ignore all others.

Definition 3. (Conflict) A conflict for $(\text{SD}, \text{COMPS}, \text{OBS})$ is a set $CS \subseteq \text{COMPS}$ such that $\text{SD} \cup \text{OBS} \cup \{\neg \text{AB}(c) \mid c \in CS\}$ is inconsistent.

A conflict corresponds to a subset of components for which it would not be consistent to assume that they all work correctly given the observations. A conflict *CS* is considered to be *minimal* if no proper subset of *CS* exists that is also a conflict.

In the original approach by Reiter the conflicts are computed through calls to a theorem prover TP. The TP component is considered to be a "black box" and no assumptions are made about how the conflicts are determined or whether they are minimal or not. In practice, however, researchers often use specific algorithms such as QUICKXPLAIN (QXP) [15], Progression [16] or MERGEXPLAIN (MXP) [17] to efficiently find the conflicts (see later sections for more details). These conflict detection algorithms, in contrast to the original assumptions by Reiter, furthermore have the advantage that they can guarantee that the returned conflict sets are minimal.

Reiter then describes the relationship between conflicts and diagnoses and shows that the set of diagnoses for a collection of minimal conflicts **CS** is equivalent to the set **H** of minimal hitting sets[1] of **CS**.

[1] Let S be a finite set and $\mathbf{C}$ be a family of subsets of S, then a subset-minimal set $H \subseteq S$ is a hitting set for $\mathbf{C}$ iff for any $C \in \mathbf{C}$ it holds that $H \cap C \neq \emptyset$. This corresponds to the set cover problem.

2.3 The Hitting Set Tree Algorithm

To determine the minimal hitting sets and therefore the diagnoses, Reiter proposes a breadth-first search procedure for the computation of a hitting set tree (HS-tree), whose construction is guided by conflicts (Algorithm 2). Furthermore, the algorithm implements different techniques to prune the search space. Algorithm 2 is sound and complete when it is guaranteed that `getConflicts` used in Algorithm 3 only returns minimal conflicts. Soundness and completeness in this context means that all returned solutions are guaranteed to be minimal diagnoses and no diagnosis for the given set of conflicts will be missed.[2]

Algorithm 2: HS-TREE ALGORITHM

Input: DPI $(\text{SD}, \text{COMPS}, \text{OBS})$
Output: All minimal diagnoses Δ

1 $\mathbf{D} \leftarrow \emptyset$; /* Initialize set of known diagnoses */
2 $\mathbf{CS} \leftarrow \emptyset$; /* Initialize set of known conflicts */
3 $frontier \leftarrow \{(\emptyset, \emptyset)\}$; /* Initialize the search frontier */
4 **while** $frontier \neq \emptyset$ **do**
  /* Get and remove the first element of the frontier */
5 $(CS, \Delta) \leftarrow \texttt{Pop}(frontier)$;
6 $frontier \leftarrow frontier \cup \texttt{processNode}(\mathbf{D}, \mathbf{CS}, (CS, \Delta))$;
7 **return D**; /* Return the set of all diagnoses */

The main principle of the HS-tree algorithm is to create a search tree where each node corresponds to a pair (CS, Δ). The first element CS represents a conflict with which a node is labeled. The second element Δ represents the set of components which are supposed to be faulty at the current node.

When the next node is retrieved from the frontier and forwarded to Algorithm 3, the set CS of the retrieved node is empty and a new label must be computed. Algorithm 3 can either reuse one of the known conflicts stored in **CS** (line 3) that is not hit by Δ or use `getConflicts` to determine one or more new conflicts (line 5). When no conflict can be reused and CS remains empty, the set Δ hits all conflicts of the given DPI and, therefore, is a diagnosis.

To guarantee the subset-minimality of the computed hitting sets, Algorithm 3 includes a pruning rule in line 2. This rule forces the algorithm to ignore all nodes where the set Δ is a superset of one of the already known minimal hitting sets.

[2] An algorithm variant called HS-DAG (Directed Acyclic Graph) is proposed in [18] for cases when the returned conflicts are not minimal.

Algorithm 3: PROCESSNODE

Input: Sets of diagnoses **D** and conflicts **CS** as well as the node (CS, Δ)
Output: A set of new nodes *frontier*

```
1  frontier ← ∅;
2  if ∀Δ' ∈ D : Δ' ⊈ Δ then                    /* If not superset of known diagnosis */
3      if ∃CS' ∈ CS : CS' ∩ Δ = ∅ then CS ← CS';          /* Reuse a conflict */
4      else                                  /* Compute a set of new conflicts not hit by Δ */
5          CS ← CS ∪ getConflicts(COMPS, SD ∪ OBS ∪ {AB(c) | c ∈ Δ});
6          CS ← Pop({CS' | CS' ∈ CS, CS' ∩ Δ = ∅}) ;      /* Retrieve a conflict */
7      if CS = ∅ then D ← D ∪ {Δ};                /* No new conflict is found */
8      else
           /* Generate successors of (CS,Δ) and add them to the frontier    */
9          frontier ← frontier ∪ {(∅, Δ ∪ {c}) | c ∈ CS};
10 return frontier
```

2.4 Example: Hitting Set Tree Construction

In the following example we show how Reiter's approach can be applied to locate a fault in a specification of a Constraint Satisfaction Problem (CSP). The example, adapted from [19], also illustrates the generality of the proposed consistency-based MBD framework.

A CSP instance I is defined as a tuple (V,D,C), where $V = \{v_1,\ldots,v_n\}$ is a set of variables, $D = \{D_1,\ldots,D_n\}$ is a set of domains for the variables in V, and $C = \{C_1,\ldots,C_k\}$ is a set of constraints. An assignment to any subset $X \subseteq V$ is a set of pairs $A = \{\langle v_1,d_1\rangle,\ldots,\langle v_m,d_m\rangle\}$ where $v_i \in X$ is a variable and $d_i \in D_i$ is a value from the domain of this variable. An assignment comprises exactly one variable-value pair for each variable in X. Each *constraint* $C_i \in C$ is defined over a list of variables S, called its scope, and forbids or allows certain simultaneous assignments to the variables in its scope. An assignment A to S *satisfies* a constraint C_i if A comprises an assignment allowed by C_i. An assignment A is a *solution* to I if it satisfies all constraints C.

Consider a CSP instance I with variables $V = \{a,b,c\}$ where each variable has the domain $\{1,2,3\}$ and where the following constraints are defined:

$$C_1 : a > b, \quad C_2 : b > c, \quad C_3 : c = a, \quad C_4 : b < c$$

Obviously, no solution for I exists and our diagnosis problem consists in finding subsets of the constraints whose definition is faulty. The engineer who has modeled the CSP could, for example, have made a mistake when writing down C_2, which should have been $b < c$. Eventually, C_4 was added later on to correct the problem, but the engineer forgot to remove C_2.

The problem can be represented as a DPI as follows by defining SD as

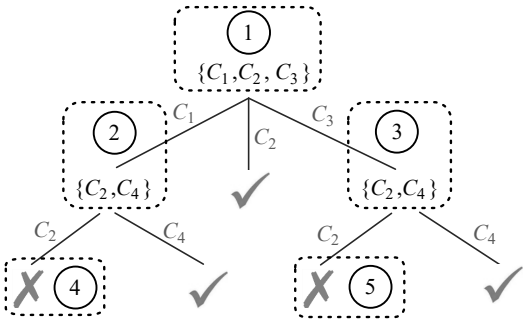

Fig. 2: Example of HS-tree construction, adapted from [19]

$$\neg\text{AB}(C_1) \rightarrow (a > b), \qquad \neg\text{AB}(C_2) \rightarrow (b > c)$$
$$\neg\text{AB}(C_3) \rightarrow (c = a), \qquad \neg\text{AB}(C_4) \rightarrow (b < c)$$

the set of components COMPS $= \{C_1, C_2, C_3, C_4\}$, and OBS $= \emptyset$.

Given the faulty definition of I, two minimal conflicts exist. Namely, the sets $\{\{C_1, C_2, C_3\}, \{C_2, C_4\}\}$ can be found by `getConflicts` using, for instance, QUICKXPLAIN as a conflict detection algorithm. Given these two conflicts, the HS-tree algorithm determines three minimal hitting sets $\{\{C_2\}, \{C_1, C_4\}, \{C_3, C_4\}\}$, which are diagnoses for the problem instance. The set of diagnoses contains the true cause of the error, the definition of C_2.

Let us now review in more detail how the HS-tree is constructed when using QUICKXPLAIN (QXP) as a conflict detection technique for the example problem. The tree construction process is illustrated in Figure 2.

Since no conflict can be reused in the first iteration, a call to QXP is made, which returns the conflict $CS = \{C_1, C_2, C_3\}$. This conflict is used to label the root node ① of the tree and is added to the set of conflicts **CS**. For each element of the conflict, a child node is created and the respective conflict element is used as a path label (line 9). Hence, in the next iteration of the main loop the frontier comprises the three nodes $(\emptyset, \{C_1\})$, $(\emptyset, \{C_2\})$, and $(\emptyset, \{C_3\})$.

Next, node $(\emptyset, \{C_1\})$ is retrieved from the frontier, shown as ② in Figure 2. Since the known conflict cannot be reused, a new conflict is computed by QXP under the assumption that C_1 is abnormal. This call returns the conflict $\{C_2, C_4\}$. The set containing only C_1 is therefore not a diagnosis and the new conflict is used as a label for node ②. The algorithm then proceeds in breadth-first style and retrieves the node $(\emptyset, \{C_2\})$. For this node none of the known conflicts can be reused and no new conflict can be found. Therefore, $\Delta = \{C_2\}$ is a diagnosis and is added to the set **D**. The corresponding node is marked with ✓ in the figure and not further expanded. At node ③, which does not correspond to a diagnosis, the already known conflict $\{C_2, C_4\}$ can be reused as it has no overlap with the node's path label. Consequently, no call to TP (QXP) is required. At the last tree level, the nodes ④ and ⑤ are not further expanded ("closed" and marked with ✗) because $\{C_2\}$ has already been

identified as a diagnosis at the previous level and the resulting diagnoses would be supersets of $\{C_2\}$. Finally, the sets $\{C_1, C_4\}$ and $\{C_3, C_4\}$ are identified as additional diagnoses.

2.5 Complexity Considerations

Finding the hitting sets for a given collection of sets – in our case a given set of conflicts – is known to be an NP-hard problem [20]. Furthermore, in many of the mentioned applications of MBD techniques to practical problems, we cannot assume that the set of minimal conflicts is given in advance. Therefore, in these applications (as well as in Algorithm 2), the conflicts are computed "on demand," i.e., during tree construction. Depending on the problem setting, finding the conflicts can be the computationally most demanding part of the entire diagnosis process.

Deciding whether an *additional diagnosis* exists when conflicts are computed on demand is also NP-complete, even for propositional Horn theories [21]. Therefore, a number of heuristics-based, approximate and thus incomplete, as well as problem-specific diagnosis algorithms have been proposed over the years. We will discuss such approaches next in Section 3.

Later on, in Section 5, however, we will focus on application scenarios where the goal is to find *all minimal diagnoses* for a given problem, i.e., we focus on *complete* algorithms. Application scenarios in which finding just one or a few diagnoses is insufficient include, for example, the MBD-based approach for finding errors in spreadsheet programs described in [6].

3 Alternative Approaches to Compute Diagnoses

Since determining diagnoses in Reiter's framework is a computationally demanding problem, a number of alternatives to the logic-based framework and to the sound and complete HS-tree diagnosis method have been proposed over the years. Three main categories of approaches can be identified: (I) approaches that limit the search to certain diagnoses, (II) approaches that use other formalisms and problem encodings than the logic-based one, and (III) approaches that trade soundness and/or completeness for efficiency.

(I) Approaches that limit the search to certain diagnoses

Approaches of this type aim to find only certain subsets of all existing diagnoses of a given DPI. The most prominent examples include approaches that (a) focus on the computation of one "minimum-cardinality" diagnosis or (b) find the k best diagnoses.

(a) Finding minimum-cardinality diagnoses conceptually means extending Reiter's definition of a diagnosis (Definition 2) with an additional requirement: Δ is a minimum-cardinality diagnosis iff there is no diagnosis Δ' such that $|\Delta| > |\Delta'|$. A minimum-cardinality diagnosis therefore comprises the smallest possible number of system components that, if assumed to be faulty, explain the observed misbehavior [22, 23, 24]. These types of diagnoses are important when analyzing very large systems for which the computation of diagnoses can be very time consuming.

(b) Approaches of this type compute k diagnoses that are optimal with respect to some predefined measure. For instance, finding the k *most probable* diagnoses is one of the most prominent examples found in the literature, see, e.g., [25, 26] or [27]. One can easily modify the HS-tree algorithm from Section 2.3 to compute the k most probable diagnoses. We only have to rewrite the condition in line 4 to $(\mathit{frontier} \neq \emptyset \wedge |\mathbf{D}| \leq k)$ and extend the function `Pop` to return the most probable node of a tree from the frontier. The latter modification turns the breadth-first scheme of the HS-tree algorithm into a uniform-cost scheme.

Parallelization approaches: Algorithms of this group can be parallelized in the same way as any uninformed search algorithm, including breadth-first or depth-first search. How these general search strategies can be parallelized is discussed in more detail in Section 4.

(II) Approaches that use alternative formalisms and problem encodings

The main idea of these approaches is to use special types of problem encodings that support a more efficient computation of the diagnoses. Siddiqi and Huang [28], for example, suggest to solve the diagnosis problem by using Bayesian networks. To speed up the computations they apply a differential approach to the reasoning in these networks [29].

Pill and Quaritsch [30], on the other hand, propose to transform a given DPI into an algebraic expression. The diagnoses are then computed by calling a recursive function H, which systematically selects a part of the provided algebraic expression and applies one of five pre-defined modification operators. One of them, for example, extracts individual elements of the selected sub-expressions that are also elements of at least one diagnosis for the given DPI.

The function H is designed in such a way that the method guarantees its termination after a finite number of rewritings. The algebraic expression returned as an output finally represents all or k diagnoses of the given DPI.

However, the currently most common and very efficient alternative way of encoding the problem is to use so-called "direct" methods, see, e.g., [23, 24, 31, 32, 33, 34]. Direct approaches transform and encode diagnosis problem instances in such a way that every model output (solution) returned by an inference engine that can process the target encoding corresponds to a diagnosis. Typical approaches encode the DPIs as constraint satisfaction (CSP) or as boolean satisfiability (SAT) problems. These

methods also support the generation of multiple diagnoses in one single call to the inference engine.

In [35], Nica et al. report the results of a series of experiments in which they compared conflict-directed search with direct encodings. Their findings indicate that for several problem settings, using the direct encoding was advantageous. However, direct methods can be applied only if there is a knowledge representation and reasoning system which is (i) expressive enough to encode the given problem setting (input system description), and in which (ii) the computation of minimal hitting sets can be embedded in some form.

In case of direct SAT encodings, one can for example weave the computation of the diagnoses into the system description [32]. Alternatively, one can reformulate the diagnosis problem as a MaxSAT problem [24]. However, there are also application areas of Model-Based Diagnosis such as the diagnosis of description logic (DL) ontologies [36], for which such direct encodings cannot be applied because existing DL reasoners are not able to generate models that correspond to diagnoses.

Parallelization approaches: The applicability of parallelization strategies for direct diagnosis methods largely depends on the approach that is used to reformulate the DPI. For instance, the methods suggested in [32] and [24] can be parallelized simply by using a SAT/MaxSAT solver that is able to compute multiple models in parallel. The parallelization of SAT and MaxSAT methods is covered in Chapter 1 on Parallel Satisfiability and Chapter 3 on Parallel Maximum Satisfiability.

(III) Approaches that trade soundness and/or completeness for efficiency

This family of methods uses "approximate" algorithms to increase the efficiency of the search process, but in exchange cannot make guarantees on soundness and/or completeness. Typical strategies include the application of stochastic search techniques such as genetic algorithms, simulated annealing, or greedy approaches.

For instance, the method presented in [37] uses a two-step greedy approach. In the first step, a random and possibly non-minimal diagnosis candidate is determined by a modified DPLL[3] algorithm. In the second step, the algorithm minimizes the candidate returned by the DPLL technique by repeatedly applying random modifications.

The approach by Li et al. [38], as another example, uses a genetic algorithm that takes a number of conflict sets as input and generates a set of bit-vectors (chromosomes), where every bit encodes a truth value of an atom over the $\text{AB}(\cdot)$ predicate. In each iteration the algorithm applies genetic operations such as mutation, crossover, etc., to obtain new chromosomes. Subsequently, all obtained bit-vectors are evaluated by a "hitting set" fitting function which eliminates bad candidates. The algorithm stops after a predefined number of iterations and returns the best diagnosis.

Parallelization approaches: So far, no proposals have been made to parallelize such approximate techniques for MBD problems. Parallel algorithms for search approaches such as simulated annealing exist [39], but it is still open so far whether

[3] Davis-Putnam-Logemann-Loveland.

these parallelization schemes can be applied to the approximate MBD algorithms described above. In particular, this is unclear because these approximate techniques usually relax the definition of the diagnosis problem in different ways.

Since no parallel versions of approximate or incomplete diagnosis algorithms have been proposed so far, the remaining sections of this chapter will focus on the parallelization of sound and complete tree-based diagnosis algorithms that can be applied to different variants of the diagnosis problem. In particular, various approaches to parallelize the HS-tree algorithm will be discussed in Section 5. It will turn out that most of these algorithms implement basic strategies from uninformed search algorithms and the difference mainly lies in the used heuristics and/or pruning rules. In the next section we will therefore first discuss general approaches for search parallelization.

4 Parallelization of Tree Search Algorithms

In this section, we summarize different general strategies to parallelize tree search processes and discuss their applicability to Model-Based Diagnosis problems.

4.1 General Parallelization Strategies

Historically, the parallelization of tree search algorithms has been approached in three different ways [40]:

(I) *Parallelization of node processing*: When applying this type of parallelization, the tree is expanded by one single process, but the computation of labels or the evaluation of heuristics is done in parallel.
(II) *Window-based processing*: In this approach, sets of nodes, called "windows," are processed by different threads in parallel. The windows are formed by the search algorithm according to some predefined criteria.
(III) *Tree decomposition approaches*: The main idea of these methods is to identify a node in a search tree such that all sub-trees originating in this node are independent and can be processed in parallel [41, 42].

In principle, all three types of parallelization can be applied in some form to the problem of generating a hitting set tree.

(I) Parallelization of node processing

Applying this strategy to the MBD problem setting means parallelizing the process of conflict computation. That is, the method `getConflicts` (Algorithm 3, line 5)

relies on a theorem prover that either makes use of multiple threads for consistency checking or that implements a parallel conflict search algorithm.

The first variant is usually simple to implement as most of the conflict computation algorithms, as in [15, 16, 43, 44], are not dependent on a particular method used in TP for consistency checking. Depending on the implemented strategy, such algorithms select a set of components $CS \subseteq \text{COMPS}$ and ask TP whether $\text{SD} \cup \text{OBS} \cup \{\neg \text{AB}(c) \mid c \in CS\}$ is inconsistent. Depending on the result, the algorithm returns a conflict set or selects another subset of components to be checked.

In the latter case – parallel conflict search – one can use, for instance, a slightly modified version of the recently proposed MERGEXPLAIN method [17], which will be discussed in more detail later in section 5.2.1. In every step, this algorithm splits the set of components into two non-empty disjoint subsets, which can then be processed in parallel. For other well-known conflict or prime implicate computation algorithms like the ones listed above, parallelizing the process is not as easy. Therefore, more research is required on parallel conflict computation when the goal is to speed up node processing through parallelization.

(II) Window-based processing

This strategy, in which multiple sets of *independent* nodes of the search tree (windows) are computed in parallel, was for example applied by Powley et al. [45]. In their work the windows are determined by different thresholds of a heuristic function of *Iterative Deepening A**.

In principle we can apply such a strategy also to the HS-tree construction problem. This would however mean that we have to categorize the nodes to be expanded according to some criterion, e.g., the probability of finding a diagnosis, and then allocate the different groups to individual threads. In the absence of such criteria, we could use a constant window size such that each open node is allocated to a separate thread. In this case the number of parallel threads (windows) should not exceed the number of physically available computing threads to obtain the best performance.

A general problem when applying a window-based strategy is that in the general case it is hard to find a window function that assigns sets of tree nodes to different threads. Ideally such a function should guarantee that the decisions made by an algorithm for nodes of different windows are independent. In case of MBD, there are two types of such decisions: labeling of nodes and pruning. For the first type, the window function has to guarantee that no two TP calls for two nodes of different windows return the same conflict. For the second type, it has to be guaranteed that the results of tree pruning for nodes of one window are irrelevant for nodes of other windows.

Unfortunately, for many MBD problem instances the computations at different levels of the tree are not independent of each other. Moreover, there is no general way to split the components of a DPI into subsets in such a way that the computation of *different* conflicts for every subset is ensured. Therefore, the parallel MBD algorithms discussed in Section 5 do not rely on window-based parallelization.

(III) Tree decomposition approaches

The idea of these approaches is to determine sub-trees in the search tree that can be processed independently of each other by different threads. Any decision made within one sub-tree must therefore not influence the behavior or decision of threads working on other parts of the search tree.

For example, Anglano et al. [46] suggest a tree decomposition approach for MBD problems in which they parallelize the diagnosis problem based on structural problem characteristics. In their work, they first map a given diagnosis problem to a Behavioral Petri Net (BPN). Then, the obtained BPN is manually partitioned into subnets and every subnet is submitted to a different Parallel Virtual Machine (PVM) for parallel processing. A major drawback of this approach is that a manual problem decomposition step is required and that such a decomposition has to be possible in the first place.

The main problem with structure-based parallelization algorithms is that they impose a number of requirements on the DPI. For example, in order to apply the decomposition approach in Binary HS-tree (BHS) algorithms [30], all conflict sets for the given diagnosis problem instance must be known in advance. Given this additional information, the method can split the search tree into two sub-trees for which the resulting sets of diagnoses are disjoint. This makes the computations in both sub-trees independent and, therefore, easy to execute in parallel. However, in the majority of cases the set of conflicts is unknown for a given DPI. The structure-based approach proposed in [47] on the other hand requires that the diagnosis problem has a tree-like structure. Overall, the applicability of tree decomposition approaches to parallel MBD is therefore limited to specific types of problem settings.

4.2 Applying Domain-Independent Parallelized Search Techniques

In principle, parallelized versions of domain-independent search algorithms such as A^* can be applied to MBD settings as well (see Chapter 11 on Parallel A* For State Space Search). However, the MBD problem has different particularities that make the application of some of these algorithms difficult. For instance, the PRA* method and its variant HDA* discussed in the work of Burns et al. [40] use a mechanism to minimize the memory requirements by retracting parts of the search tree. These "forgotten" parts are later regenerated when required. In our MBD setting, the generation of nodes is however the most costly part, which is why the applicability of HDA* seems limited. Similarly, duplicate detection algorithms such as PBNF [40] require the existence of an abstraction function that partitions the original search space into blocks. In general MBD settings, however, we cannot assume that such a function is given.

In order to improve the performance of a parallel algorithm one should in general try to avoid the duplicate generation of nodes by different threads. A promising starting point for this research could be the work by Phillips et al. [48]. The authors

suggest a variant of the A* algorithm that generates only independent nodes in order to reduce the costs of node generation. Two nodes are considered to be independent if the generation of one node does not lead to a change of the value returned by the heuristic function for the other node. The generation of independent nodes can then be done in parallel without the risk of the repeated generation of an already known state. The main difficulty when adopting this algorithm for MBD is the formulation of an admissible heuristic required to evaluate the independence of the nodes for arbitrary diagnosis problems. However, for specific problems that can be encoded as CSPs, Williams and Ragno [26] present a heuristic that depends on the number of unassigned variables at a particular search node.

Finally, parallelization has also been used in the literature to speed up the search in very large search trees that do not fit in memory. Korf [49], for instance, suggests an extension of a hash-based delayed duplicate detection algorithm that allows a search algorithm to continue search while other parts of the search tree are written to or read from the hard drive. Such methods can in theory be used in combination with parallel MBD algorithms in case of complex diagnosis problems.

5 Parallelized Hitting Set Tree Construction Schemes

In this section, we review two algorithms that implement parallelization strategies for Reiter's sound and complete HS-tree algorithm. As discussed in Section 2.3, in every iteration the HS-tree algorithm picks a node from the queue, labels it with a conflict set, generates a set of successor nodes, and applies the pruning rules. The two approaches to parallelize this process considered in this section follow the general idea of breath-first search parallelization. In particular, they assign every iteration step of the main loop to a different thread. The main problem of such a scheme, however, is that the generation of node labels, i.e., the computation of conflict sets, can be time consuming. Therefore, we also show three possible extensions to these schemes which move more of the processing power to the conflict computation task.

5.1 Computing Multiple Hitting Set Tree Nodes in Parallel

The two algorithms presented next aim to maintain the breadth-first tree exploration scheme of the HS-tree algorithm and implement strategies that utilize multiple threads without sacrificing the soundness and completeness of the diagnosis process. Furthermore, both algorithms do not make any assumptions about specific properties of the diagnosis problem instances to be solved. Therefore, they can be applied to different variations of the diagnosis problem definition from Section 2. Conceptually, the presented algorithms can also be seen as special cases of the window-based parallelization scheme described in Section 4.1, where every window comprises exactly one node.

5.1.1 Level-Wise Parallelization

The strategy of the first parallelization scheme is to examine all nodes *of one tree level* in parallel and to proceed with the next level only when all elements of the current level have been processed.

In the example shown in Figure 2, this would mean that the computations (TP calls) required for the three first-level nodes labeled with $\{C_1\}$, $\{C_2\}$, and $\{C_3\}$ can be processed in three parallel threads. The nodes of the next level are processed when all threads of the previous level are finished.

Using this Level-Wise Parallelization (LWP) scheme, the breadth-first order is strictly maintained. The parallelization of the computations is generally feasible because the consistency checks for each node can be done independently of those done for the other nodes on the same level. Synchronization is only required to make sure that no thread starts exploring a path that is already under examination by another thread.

Algorithm 4: DIAGNOSELW: Level-Wise Parallelization

```
   Input: DPI (SD, COMPS, OBS)
   Output: All minimal diagnoses Δ
1  D ← ∅ ;                                  /* Initialize set of known diagnoses */
2  CS ← ∅ ;                                 /* Initialize set of known conflicts */
3  frontier ← {(∅,∅)} ;                     /* Initialize the search frontier */
4  level ← ∅;                               /* Initialize level-wise processing */
5  while frontier ≠ ∅ do
       /* Get and remove the first element of the frontier                  */
6      (CS,Δ) ← Pop(frontier);
7      level ← level ∪ runParallel({processNode(D,CS,(CS,Δ))});
8  wait();
9  if level ≠ ∅ then frontier ← level; goto 4;
10 return D;                                /* Return the set of all diagnoses */
```

Algorithm 4 shows how the sequential method implemented in Algorithm 2 can be adapted to support this parallelization approach.[4] The statement `runParallel` takes a function as a parameter and schedules it for execution in a pool of threads of a given size. With a thread pool of, e.g., size 2, the generation of the first two nodes is done in parallel and then the main thread waits until one of the threads finishes. Only then will the third node be processed. Using this mechanism we can ensure that the number of threads executed in parallel is less than or equal to the number of hardware threads or CPUs.

In addition, during the execution of the algorithm all changes to global structures such as **D** and **CS** have to be synchronized. That is, two threads may read the structures concurrently, but writing to one of these data structures is exclusive. While one thread is writing, all other threads must wait until the operation has finished

[4] Differences to the original Algorithm 2 are highlighted with a gray background.

regardless of whether they want to read or write.[5] Finally, the statement `wait` is used for synchronization and blocks the execution of the subsequent code until all scheduled threads are finished.

Theorem 1 ([19]). *Level-Wise Parallelization is sound and complete.*

5.1.2 Full Parallelization

The LWP scheme maintains the breadth-first order of the original algorithm and, therefore, inherits all its properties, such as soundness and completeness. However, in some situations the level-wise processing procedure might get stuck at the end of a level when the computation of a conflict for one of the nodes takes a long time. In this case, all threads that have already finished processing their nodes have to wait for the one thread still working.

The Full Parallelization (FP) approach presented in Algorithm 5 immediately schedules the first node of the frontier for execution as soon as some thread becomes idle. In this way, the FP scheme avoids the problem observed for the LWP procedure. The main loop continues until the frontier is empty and no more nodes are processed in parallel.

Algorithm 5: DIAGNOSEFP: Full Parallelization

```
   Input: DPI (SD, COMPS, OBS)
   Output: All minimal diagnoses Δ
1  D ← ∅ ;                                  /* Initialize set of known diagnoses */
2  CS ← ∅ ;                                 /* Initialize set of known conflicts */
3  frontier ← {(∅,∅)} ;                     /* Initialize the search frontier */
4  while frontier ≠ ∅ ∨ runningThreads > 0 do
       /* Get and remove the first element of the frontier                  */
5      (CS, Δ) ← Pop(frontier);
6      frontier ← frontier ∪ runParallel({processNode(D, CS, (CS, Δ))});
7      checkMinimality(D);
8  return D;                                /* Return the set of all diagnoses */
```

Since the nodes are expanded as soon as possible, FP might not follow the breadth-first strategy of the HS-tree algorithm anymore. Consequently, it may find non-minimal diagnoses during the search. Therefore, in every iteration the elements stored in the set of known diagnoses $\mathbf{D}$ must be checked for minimality. The method `checkMinimality` removes all Δ_j from the set $\mathbf{D}$ if there exists $\Delta_i \in \mathbf{D}$ such that $\Delta_i \subset \Delta_j$. Note that the check for minimality has to be synchronized on $\mathbf{D}$. That is, all other threads are not allowed to modify $\mathbf{D}$ while its elements are removed.

From the performance perspective, the FP method ensures that all its threads are busy as long as there is at least one node in the frontier. On the other hand it needs

[5] Implementing such concurrency aspects is comparatively simple in modern programming languages such as Java, e.g., by using the `synchronized` keyword.

more time to synchronize the threads and to remove redundant hitting sets. If the computation of conflicts does not take long and the last nodes of a level are finished simultaneously, then LWP can be advantageous. These aspects will be discussed in more detail in Section 6 where we present results of an empirical comparison of FP and LWP for different problems.

Theorem 2 ([19]). *Full Parallelization is sound and complete, if applied to find all diagnoses up to some cardinality.*

Theorem 3 ([19]). *Full Parallelization cannot guarantee completeness and soundness when applied to find the first k diagnoses, i.e.,* $1 \leq k < N$*, where N is the total number of diagnoses of a problem.*

Note that in cases when FP is used to search for only k diagnoses, every computed hitting set must additionally be minimized by applying an algorithm such as INV-QUICKXPLAIN [50]. Similarly to QUICKXPLAIN, INV-QUICKXPLAIN applies a divide-and-conquer strategy, but in this case to find one minimal diagnosis for a given diagnosis problem instance. Applied to a given, possibly non-minimal hitting set H, this algorithm can find a minimal hitting set $H' \subseteq H$ requiring only $O(|H'| + |H'| \log(|H|/|H'|))$ calls to the theorem prover TP. The first part, $|H'|$, corresponds to the number of TP calls required to determine whether or not H' is minimal. The second part indicates the number of subproblems that must be considered by INV-QUICKXPLAIN's divide-and-conquer strategy to find the minimal hitting set H'.

5.2 Computing Nodes and Conflicts in Parallel

In all versions of the HS-tree algorithm presented above, the TP call (`getConflicts`) corresponds to an invocation of QXP. Whenever a new node of the HS-tree is created, QXP returns a set of elements that represents exactly one new conflict. This strategy has the advantage that the call to TP immediately returns after one conflict has been determined. This in turn means that the other parallel execution threads immediately "see" this new conflict in the shared data structures and can, in the best case, reuse it when constructing new nodes.

A disadvantage of computing only one conflict at a time with QXP is that the search for conflicts is *restarted* on each invocation. A recently proposed new conflict detection technique called MERGEXPLAIN (MXP) [17] is capable of computing multiple conflicts in one call. The general idea of MXP is to continue the search after the identification of the first conflict and look for additional conflicts in the remaining constraints (or logical sentences) in a divide-and-conquer approach.

When combined with a sequential HS-tree algorithm, the effect is that during tree construction more time is initially spent on conflict detection before the construction continues with the next node. In exchange, the chances of having a conflict available for reuse increase for the next nodes. At the same time, the identification of some of

the conflicts is less time-intensive as smaller sets of constraints have to be investigated due to the divide-and-conquer approach of MXP. An experimental evaluation on various benchmark problems shows that substantial performance improvements are possible in a sequential HS-tree scenario when the goal is to find *a few leading diagnoses* [17].

5.2.1 Background: QUICKXPLAIN and MERGEXPLAIN

The QXP method implemented in Algorithm 6 is a conflict detection technique which was originally applied to find one minimal conflict set for a set of unsatisfiable constraints.[6] However, over the last decade it was often applied to find conflicts for Model-Based Diagnosis problems.

QXP implements a recursive divide-and-conquer strategy that operates on two sets of constraints B and C. The set B – called "background theory" – comprises all constraints that are considered to be correct in the current recursive call. When QXP starts, this set is initialized with SD, OBS, and $\{\neg \text{AB}(c) \mid c \in L\}$, where L is a set of components used as arc labels on the path from the root to the current node in the HS-tree (*visited nodes*). The set C comprises all constraints that are possibly faulty and in which we search for a conflict.

In case the set C is consistent with the background theory or C is empty, then no conflict can be found and the algorithm immediately returns. Otherwise, QXP calls `computeConflict`, which corresponds to Junker's QUICKXPLAIN' function in [15]. The only difference between these methods is that `computeConflict` does not require a strict partial order for the set of constraints C. We omit the requirement for a strict partial order here, as in many applications of MBD prior information about fault probabilities is not available.

Roughly, QXP applies a divide-and-conquer strategy that has two modes: "search" and "extraction." The algorithm starts in the search mode, in which every recursive call partitions the set of faulty constraints C into two sets C_1 and C_2. If C_1 is inconsistent, then QXP will continue to search for a conflict within this set and partitions C_1 in the next recursive call. The algorithm switches into the extraction mode if the partitioning process has split all conflicts of C into two parts, i.e., C_1 is consistent. In this mode QXP finds the first part of a conflict in the set C_2 and then the second part in the set C_1.

The recently proposed MXP algorithm extends the ideas of QXP by searching for conflicts not only in C_1, but also in C_2 in one call. This results in the computation of multiple conflicts, if they exist. Algorithm 7 presents the general procedure of MXP. First, the algorithm checks whether the sets of input constraints actually include at least one conflict. Next, it calls the function `findConflicts`, which returns a tuple $\langle \mathsf{C}', \mathbf{CS} \rangle$, where C' is a set of remaining consistent constraints and **CS** is a set of

[6] Hereafter we use the term constraints as it was done in the original paper [15]. However, note that QXP uses the theorem prover only for consistency checking and is independent of the underlying reasoning technique. Therefore, the elements of the sets could be any set of logic sentences for which sound and complete reasoning methods exist.

Algorithm 6: QUICKXPLAIN (QXP)

Input: A diagnosis problem (SD, COMPS, OBS), a set *pathNodes* of labels on the path from the root to the current node
Output: A set containing one minimal conflict $CS \subseteq \mathsf{C}$

```
1  B = SD ∪ OBS ∪ {AB(c) | c ∈ pathNodes};
2  C = {¬AB(c) | c ∈ COMPS \ pathNodes};
3  if isConsistent(B ∪ C) then return 'no conflict';
4  else if C = ∅ then return ∅;
5  return {c | ¬AB(c) ∈ computeConflict(B, B, C)};

   function computeConflict(B, D, C)
6      if D ≠ ∅ ∧ ¬isConsistent(B) then return ∅;
7      if |C| = 1 then return C;
8      Split C into disjoint, non-empty sets C1 and C2
9      D2 ← computeConflict(B ∪ C1, C1, C2)
10     D1 ← computeConflict(B ∪ D2, D2, C1)
11     return D1 ∪ D2;
```

found conflicts. Similarly to QXP's `computeConflict` this function first recursively partitions the set C into two subsets. However, in contrast to `computeConflict`, `findConflicts` continues the search for conflicts in both subsets. This allows the algorithm to identify conflict sets in both C_1 and C_2. Every found conflict is stored in the set **CS** and is resolved by removing one of its elements from the set C_1. Finally, after all conflicts in the subsets C_1 and C_2 have been found and resolved, the function checks whether the union of these two sets is consistent. If not, it searches for a conflict in $\mathsf{C}'_1 \cup \mathsf{C}'_2$ (and the background theory) in the style of QXP.[7]

5.2.2 Strategies for Combining Node and Conflict Computation

The main idea of the following strategies is to invest more processing power in the task of conflict computation while the nodes of the HS-tree are constructed in parallel using LWP or FP. Our expectation is that higher conflict reuse levels can be achieved during tree construction as we know more conflicts at the beginning of the process.

The desired effect can be achieved by embedding variants of MXP as a conflict detection strategy, because in MXP we invest more time in looking for *additional* conflicts in one call before we proceed with the next node. In principle, computing one conflict with MXP requires at least one consistency check more for every partition than QXP. However, MXP should still be advantageous because it can avoid the potential overheads that can happen when the same conflict is computed simultaneously by LWP and FP.

In the following we discuss different ways of incorporating MXP within the full parallelization scheme FP:

[7] Please see [17] for more details. The paper also contains the results of an in-depth experimental analysis for different problems.

Algorithm 7: MERGEXPLAIN (MXP)

Input: A diagnosis problem (SD, COMPS, OBS), a set *pathNodes* of labels on the path from the root to the current node
Output: **CS**, a *set* of minimal conflicts

```
1  B = SD ∪ OBS ∪ {AB(c) | c ∈ pathNodes};
2  C = {¬AB(c) | c ∈ COMPS \ pathNodes};
3  if ¬isConsistent(B) then return 'no solution';
4  if isConsistent(B ∪ C) then return ∅;
5  ⟨_, CS⟩ ← findConflicts(B, C)
6  return {c | ¬AB(c) ∈ CS};

   function findConflicts(B, C) returns tuple ⟨C′, CS⟩
7      if isConsistent(B ∪ C) then return ⟨C, ∅⟩;
8      if |C| = 1 then return ⟨∅, {C}⟩;
9      Split C into disjoint, non-empty sets C1 and C2
10     ⟨C′1, CS1⟩ ← findConflicts(B, C1)
11     ⟨C′2, CS2⟩ ← findConflicts(B, C2)
12     CS ← CS1 ∪ CS2;
13     while ¬isConsistent(C′1 ∪ C′2 ∪ B) do
14         X ← computeConflict(B ∪ C′2, C′2, C′1)
15         CS ← X ∪ computeConflict(B ∪ X, X, C′2)
16         C′1 ← C′1 \ {α} where α ∈ X
17         CS ← CS ∪ {CS}
18     return ⟨C′1 ∪ C′2, CS⟩;
```

Strategy (1)

One first strategy is simply to call MXP instead of QXP during node generation. Whenever MXP finds a conflict, it is added to the global list of known conflicts and can be (re-)used by other parallel threads. The thread that executes MXP during node generation continues with the next node when MXP returns.

Strategy (2)

This strategy implements a variant of MXP that is slightly more complex. Once MXP finds the first conflict, the method immediately returns this conflict so that the calling thread can continue exploring additional nodes. At the same time, a new background thread is started which continues the search for additional conflicts, i.e., it completes the work of the MXP call. In addition, whenever MXP finds a new conflict it checks whether any other already running node generation thread could have reused the conflict if it had been available beforehand. If this is the case, the search for conflicts in this *other* thread is stopped as no new conflict is needed anymore. Strategy (2) could in theory result in better CPU utilization, as we do not have to wait for an MXP call to finish before we can continue building the HS-tree. However, the strategy also leads to higher synchronization costs between

the threads as, for instance, we have to potentially notify the working threads about newly identified conflicts.

Strategy (3)

A final strategy is to parallelize the conflict detection procedure of MXP itself. Whenever the set C of constraints is split into two parts, the first recursive call of `findConflicts` is queued for execution in a thread pool and the second call is executed in the current thread. When both calls are finished, the algorithm can continue. An empirical evaluation of this approach in [19] however indicated that the additional gains that can be obtained through this strategy are limited. Nonetheless, parallelizing the conflict detection algorithm – which can be any sort of Theorem Prover – in principle represents a possible strategy to speed up the overall tree construction process.

6 Effectiveness of Computing Multiple Nodes in Parallel

The goal of this section is to quantify the possible gains that can be obtained through parallelization for typical Model-Based Diagnosis problems. In this section, we will analyze the effectiveness of the two approaches presented in Section 5.1 to parallelize Reiter's HS-tree algorithm (Level-Wise Parallelization and Full Parallelization) as examples. The presented results are taken from [19], which also contains a detailed discussion of a number of additional experiments.

6.1 General Considerations

In our experiments, we will use *wall times* as a basic measure for our evaluation, because the comparison of wall times is a common approach in the literature to assess the improvements that one can obtain through parallelization. Wall times represent a start-to-end measurement approach for a given task, which means that also times are included when processors have to wait for resources etc. Using wall time instead of CPU time is particularly appropriate for the given problem setting, because the synchronization between threads in particular in the FP algorithm can take a significant amount of time.

The differences in the wall times are often reported with the help of two measures, *speedup* and *efficiency*, that take the amount of available computing resources into account. Speedup S_p is computed as $S_p = T_1/T_p$, where T_1 is the wall time when using one thread (the sequential algorithm) and T_p the wall time when p parallel threads are used. The efficiency E_p is defined as S_p/p and compares the speedup with the theoretical optimum.

While speedup and efficiency are well defined, one has to be careful when interpreting or trying to generalize the results, because the speedups that can be achieved depend not only on the algorithms, but can also be influenced by the underlying hardware architecture. With Intel's Hyper-Threading Technology, for example, virtual computing cores can be used, which are, however, mapped to the same physical cores. For a parallel program, these virtual cores appear like real physical computing nodes, but the results that are obtained when using virtual cores can be different from those that one would see with more physical cores. In addition, an algorithm can perform differently when executed on a single CPU with multiple cores or on a server architecture with multiple CPUs on a single main board. Furthermore, running the same algorithm on multiple computers connected in a network might lead to yet different results. The results that are reported below were obtained with specific hardware configurations. For alternative hardware architectures, the speedups might be different and other algorithms might even be better suited.

Besides the hardware on which the experiments are executed, also the choice of the benchmark problems influences the obtained results. Obviously, problems that we only need a few milliseconds to solve in a single-threaded system are not a good subject to study the possible benefits of parallelization. In such cases, the speedups that might eventually be achieved can easily be eaten up by the overhead costs of parallelization. Starting a new execution thread in Java, for example, is often considered expensive, e.g., due to the costs of thread initialization and lifecycle management. In addition to the general complexity of the individual benchmark problems, we should furthermore also look at certain other characteristics of the individual problems that might impact the benefits of parallelization. Which of the characteristics are relevant, however, might depend on the specific parallelization algorithm. For example, the average width of the search tree can impact the performance of the level-wise approach LWP, i.e., if the tree is not very wide, the degree of parallelization that can be achieved will be limited.

Overall, the discussions show that a number of aspects, both hardware-related ones and problem-specific ones, can impact the effectiveness of different parallelization strategies. In [19], the proposed parallelization approaches were therefore tested on a variety of different benchmark problems, and we will summarize some of the results next. The experiments were limited to two different types of standard hardware architectures and did not require any special computing equipment for massive parallelization or the utilization of the processing power of Graphics Processing Units (GPUs). The obtained results will show that even with standard hardware and general-purpose programming languages significant speedups can be achieved.

6.2 Results for Standard Electronic Circuit Benchmark Problems

In order to evaluate the usefulness of the LWP and FP parallelization strategies in comparison with Reiter's original single-threaded method, a number of diagnosis problems from three different application domains were used in [19]. In this chap-

ter we report the detailed results of one of these domains, the electronic circuit benchmarks from the DX Competition 2011 Synthetic Track [51], and summarize the overall findings. The detailed results for the other problem domains – faulty descriptions of Constraint Satisfaction Problems (CSPs) as well as problems from the domain of ontology debugging – can be found in [19].

For the evaluation on the DX benchmarks the first five systems of the competition dataset were used (see Table 1). For each system, the competition specifies 20 scenarios with injected faults that result in different faulty output values. The system descriptions and the given input and output values were used for the diagnosis process, while the additional information about the injected faults was of course ignored. The problems were converted into Constraint Satisfaction Problems, which allowed us to simulate the behavior of the circuits. In the experiments Choco [52] served as a constraint solver and QXP was used for conflict detection.

System	#C	#V	#F	#D	$\overline{\#D}$	$\overline{\lvert D \rvert}$
74182	21	28	4 - 5	30 - 300	139.0	4.66
74L85	35	44	1 - 3	1 - 215	66.4	3.13
74283*	38	45	2 - 4	180 - 4,991	1,232.7	4.42
74181*	67	79	3 - 6	10 - 3,828	877.8	4.53
c432*	162	196	2 - 5	1 - 6,944	1,069.3	3.38

Table 1: Characteristics of the selected DXC benchmarks

Table 1 shows the characteristics of the systems in terms of the number of constraints (#C) and the problem variables (#V).[8] The number of injected faults (#F) and the number of calculated diagnoses (#D) vary strongly because of the different scenarios for each system. For both columns we show the ranges of values over all scenarios. The columns $\overline{\#D}$ and $\overline{|D|}$ indicate the average number of diagnoses and their average cardinality. As can be seen, the search tree for the diagnosis can become extremely broad with up to 6,944 diagnoses with an average diagnosis size of only 3.38 for the system c432.

System	**Seq.**	**LWP**		**FP**	
	[ms]	$\mathbf{S}_4$	$\mathbf{E}_4$	$\mathbf{S}_4$	$\mathbf{E}_4$
74182	65	2.23	0.56	**2.28**	**0.57**
74L85	209	2.55	0.64	**2.77**	**0.69**
74283*	371	2.53	0.63	**2.66**	**0.67**
74181*	21,695	1.22	0.31	**3.19**	**0.80**
c432*	85,024	1.47	0.37	**3.75**	**0.94**

Table 2: Observed performance gains for the DXC benchmarks

[8] For systems marked with *, the search depth was limited to the actual number of faults to ensure that the sequential algorithm terminated within a reasonable time frame.

Table 2 shows the averaged results when searching for all minimal diagnoses in the DXC benchmarks. We first list the running times in milliseconds for the sequential version (Seq.) and then the improvements of LWP or FP in terms of speedup $\mathbf{S}_4$ and efficiency $\mathbf{E}_4$ with respect to the sequential version. The fastest algorithm for each system is highlighted in bold.

In all tests, both parallelization approaches outperform the sequential algorithm. Furthermore, the difference between the sequential algorithm and one of the parallel approaches was statistically significant ($p < 0.05$) in 95 of the 100 tested scenarios. For all systems, FP was more efficient than LWP and the speedups range from 2.28 to 3.75 (i.e., up to a reduction in running time of more than 70%). In 59 of the 100 scenarios the difference between LWP and FP was statistically significant. A trend that can be observed is that the efficiency of FP was higher for the more complex problems. The reason is that for these problems the time needed for node generation is much larger in absolute numbers than the additional overhead times that are required for thread synchronization.

As mentioned above, additional experiments for other problem domains were reported in [19]. The obtained results show that parallelizing the HS-tree algorithm is also advantageous for these domains. For the CSP and ontology debugging problems, however, FP was not consistently faster than LWP. This indicates that the advantage of FP over LWP can depend on the characteristics of the problems. In addition, for some scenarios the speedups of the parallelized approaches were not as high as the speedups achieved for the DXC benchmark problems.

6.3 Systematic Variation of Problem Characteristics

The empirical analyses reported in the previous section for typical MBD benchmark problems show that computing multiple nodes in parallel can help to significantly speed up the diagnosis process. Both parallelization techniques were faster than the sequential algorithm in all tests. However, the full parallelization approach FP was not consistently faster than the level-wise approach LWP across all tested problem instances.

6.3.1 Method

To obtain a better understanding of how different problem characteristics impact the performance of the parallelization techniques, a series of additional experiments with synthetic problem instances was performed in [19]. For these experiments, a suite of hitting set computation problems was created where the following characteristics were systematically varied: number of components (#Cp), number of conflicts (#Cf), and average size of the conflicts ($\overline{|\text{Cf}|}$).

To create these diagnosis problem instances, a problem generation algorithm was designed which constructs a set of minimal conflicts of the specified average

size for the given number of components. To obtain realistic scenarios, not all generated conflicts were of equal size but their size was varied in a randomized process according to a Gaussian distribution with the desired size as a mean. Similarly, since not all components should be equally likely to be part of a conflict, again a randomized process was used to assign failure probabilities to the components.

An additional aspect that can impact the performance of the different parallelization techniques is the time that is required to compute one conflict to label a new node in the search tree. For the suite of synthetic benchmark problems, the conflicts are known in advance (as they were used to construct the diagnosis problems in the first place). A call to the theorem prover would therefore simply mean looking up one of the known conflicts from memory, which requires almost no computation time. To be still able to measure the impact of varying conflict computation times, artificial processing delays were introduced into the diagnosis process to simulate varying conflict detection times. Technically, this can be done by adding artificial and slightly randomized *waiting times* (Wt) upon each consistency check inside the theorem prover. Of course, the consistency-checking method is only called if no already retrieved conflict can be reused for the current node.

#Cp, #Cf, $\overline{\lvert Cf \rvert}$	#D	Wt [ms]	**Seq.** [ms]	**LWP** $\mathbf{S}_4$	**LWP** $\mathbf{E}_4$	**FP** $\mathbf{S}_4$	**FP** $\mathbf{E}_4$
Varying computation times Wt							
50, 5, 4	25	**0**	23	2.26	0.56	**2.58**	**0.64**
50, 5, 4	25	**10**	483	2.98	0.75	**3.10**	**0.77**
50, 5, 4	25	**100**	3,223	2.83	0.71	**2.83**	**0.71**
Varying conflict sizes							
50, 5, **6**	99	10	1,672	3.62	0.91	**3.68**	**0.92**
50, 5, **9**	214	10	3,531	3.80	0.95	**3.83**	**0.96**
50, 5, **12**	278	10	4,605	3.83	0.96	**3.88**	**0.97**
Varying numbers of components							
50, 10, 9	201	10	3,516	**3.79**	**0.95**	3.77	0.94
75, 10, 9	105	10	2,223	**3.52**	**0.88**	3.29	0.82
100, 10, 9	97	10	2,419	3.13	0.78	**3.45**	**0.86**
#Cp, #Cf, $\overline{\lvert Cf \rvert}$	#D	Wt [ms]	**Seq.** [ms]	**LWP** $\mathbf{S}_8$	**LWP** $\mathbf{E}_8$	**FP** $\mathbf{S}_8$	**FP** $\mathbf{E}_8$
Adding more threads (8 instead of 4)							
50, 5, **6**	99	10	1,672	6.40	0.80	**6.50**	**0.81**
50, 5, **9**	214	10	3,531	7.10	0.89	**7.15**	**0.89**
50, 5, **12**	278	10	4,605	7.25	0.91	**7.27**	**0.91**

Table 3: Simulation results

6.3.2 Results

The results of the systematic variation of the problem characteristics are shown in Table 3. The table shows the effects of varying the conflict computation times, effects of different conflict sizes, effects of different problem sizes in terms of system components, and finally the effects of using more parallel computation threads.

The following observations can be made. First, if conflicts can be found in almost no time ($\mathrm{Wt} = 0$) parallelizing the computation of multiple nodes still helps to speed up the overall diagnosis process, but due to the overhead of thread creation and synchronization the benefits of parallelization are greater for cases in which the conflict computation actually takes at least a few milliseconds.

Second, larger conflicts ($\overline{|\mathrm{Cf}|}$) and correspondingly broader HS-trees are better suited for parallel processing. On the other hand, increasing the number of components with an unchanged number and size of conflicts leads to larger diagnoses. Searching for diagnoses up to a pre-defined search depth in this case leads to fewer found diagnoses and a narrower search tree. As a result, the relative performance gains of the parallelized algorithms are lower than when there are fewer components. Finally, when there are larger conflicts, using more threads leads to further improvements as in these cases even higher levels of parallelization can be achieved.

Overall, the simulation experiments clearly demonstrate that parallelization is advantageous for a variety of problem configurations. For all tests, the speedups of LWP and FP are statistically significant. The results also reveal how the different problem characteristics of the underlying problem impact the possible performance gains. Finally, regarding the comparison of the LWP and the FP scheme, the additional gains of not waiting at the end of each search level (as done by the FP method) typically led to small further improvements.

7 Alternative Model-Based Diagnosis Parallelization Approaches

The parallelization approaches presented in the previous sections maintain the generic and problem-independent nature of the original HS-tree algorithm. With slight and straightforward adaptations they are furthermore applicable to different variations of the general definition of the diagnosis problem introduced in Section 2. In this section we briefly review existing alternative parallelization approaches from the recent literature that were developed for specific diagnosis problem settings.

7.1 Tree-Based Approaches To Find One or Few Diagnoses

The first situation we consider here is when computing all minimal diagnoses is extremely challenging in cases when, e.g., a system to be diagnosed is huge or the computation of even one minimal conflict is too complex and takes unacceptably

long. Also, there could be application scenarios where the allowed response time to return the diagnoses is very limited. In such settings, one can try to focus on a specific subset of all existing diagnoses which might be easier to compute and, e.g., search for a predefined number of diagnoses or for diagnoses of a limited cardinality (some examples are given in Section 3).

Different heuristic, stochastic, or approximative algorithms have been suggested for such situations in the literature [22, 32, 37]. The internal designs of these approaches are quite diverse. Therefore, it is challenging to analyze them with respect to the potential benefits of parallelization in a general manner.

As a simplification and approximation of such algorithms, one can however look at the possible benefits of parallelizing a depth-first strategy to find a limited number of diagnoses. If parallelizing such a strategy proves beneficial, we can see this as an indicator that parallelizing other strategies such as those mentioned above could be worth investigating. In [19], the results of a number of experiments are reported in which two variants of such tree-based approaches were compared with the Full Parallelization approach from Section 5.

Parallel Random Depth-First Search (PRDFS)

The PRDFS method aims at the fast computation of *one single diagnosis*, using a depth-first strategy of expanding the search tree. Given the root node of the search tree, every parallel execution thread of the algorithm greedily searches for a diagnosis by expanding random nodes of the tree depth-first. Whenever a node has been labeled with a conflict, each PRDFS thread – in contrast to the HS-tree algorithm – randomly selects one component of the conflict and generates only one successor. This node is then expanded in the next iteration of the thread's main loop.

Whenever a diagnosis is found with this greedy strategy, it is obviously not guaranteed that the diagnosis is minimal. As in the situation when applying the FP strategy to search for a limited number of diagnoses, every returned diagnosis therefore has to be minimized, i.e., the redundant elements have to be removed.

A Hybrid Strategy

This algorithm, similar to PRDFS, focuses on the computation of one single diagnosis. The algorithm however considers that depending on the cardinality of the existing minimal diagnoses it can either be advantageous to quickly descend in the search tree or to exhaustively look for diagnoses of very small sizes first. In the proposed hybrid strategy, half of the available threads therefore follow the PRDFS scheme to descend in the search tree and the other half of them explore the tree in breadth-first manner using the FP algorithm.

The coordination between the two algorithms can be done with the help of shared data structures that contain the known conflicts and diagnoses. When enough

diagnoses (e.g., one) are found, all running threads can be terminated and the results are returned.

Insights

In the experiments from [19], the same benchmark problems were used as for the evaluation of the LWP and FP techniques. The goal this time however was to find one arbitrary minimal diagnosis. The obtained results can be summarized as follows. When only one diagnosis is needed, using a depth-first strategy is in most cases faster than using the FP technique and thus faster than Reiter's single-threaded HS-tree algorithm. Using multiple threads in this depth-first search (PRDFS) is in almost all cases beneficial. Finally, the hybrid strategy represents a good compromise whose performance on average is between the breadth-first FP scheme and the PRDFS method.

7.2 Distributed Hitting Set Algorithms with Known Conflicts

Several research papers in the MBD literature make the assumption that the set of (non-minimal) conflicts is already given at the beginning of the diagnosis process. Consequently, the diagnosis problem is reduced to the construction of hitting sets.

For such situations, Cardoso and Abreu in [53] suggested a distributed version of their STACCATO algorithm [54], which is based on the popular MapReduce computation scheme [55]. The proposed algorithm computes the minimal hitting sets, and thus the minimal diagnoses, in a distributed manner, given the (non-minimal) set of conflicts as an input.

In every execution step, the algorithm builds a hitting set d' by adding a component to it that hits at least one of the so far *unhit* minimal conflict sets. The selection of the component is done from a queue R that comprises all components that are not in d'. In addition, the elements of R are ranked according to the Ochiai coefficient.

The mapping step implements two functions that assign elements of the queue to one of the n available processes. The mapping can be done with one of two possible functions, *stride* and *random*. The first function assigns elements in a cyclical manner, i.e., process k gets a component l if $(l \mod n) = k-1$. The second function assigns components by randomly sampling from a uniform distribution.

The authors compared their distributed version of the algorithm with the single-threaded STACCATO method using a variety of artificially created benchmark problems. Their analyses showed that the new algorithm is faster than the previous one in both a distributed and a single-CPU setup. Furthermore, the required additional overhead for distributing the problem across computing nodes seemed to diminish in particular for the large problem instances.

In [56], Zhao and Ouyang suggest two further algorithms that can be used in distributed settings. Given a set of conflicts, the first algorithm starts with the par-

titioning of this set such that any two conflicts from different partitions share no components. Next, it computes minimal hitting sets for every partition and finds the set of diagnoses. To compute a diagnosis in this set, the algorithm selects one minimal hitting set for every partition and then joins them.

The second proposed algorithm is used in cases when additional conflicts arise after the diagnoses are already computed. The main idea is to update the families of conflict sets with new elements and find diagnoses in a distributed way, which is done in a similar way to the first algorithm.

The parallelization approach in this work relies on the fact that hitting sets of such partitions can be computed in parallel. This resulted in a considerable reduction of the required computation times compared to the single-threaded Boolean [57] and Boolean-HS-Tree [30] algorithms.

8 Summary

The computation of possible explanations of an unexpected system behavior using Model-Based Diagnosis approaches can be computationally challenging, in particular in application scenarios in which it is not sufficient to know only *some* heuristically determined diagnoses.

Even though multi-core computers are common today, and in different domains relying on the processing power of Graphics Processing Units has proven to be useful, limited research exists so far on parallel computation approaches in the context of Model-Based Diagnosis.

In this chapter, we have focused on different strategies for parallelizing Reiter's classical HS-tree algorithm on multi-core computers. While the algorithm in principle follows a breadth-first tree search strategy, using the concept of conflicts is essential to prune the search space, which is why the presented parallelization approaches try to maintain the basic character of the algorithm.

We believe that the presented algorithms therefore only represent a first step toward a better usage of the computing power that we have available today. Instead of being parallel versions of existing single-threaded algorithms, future Model-Based Diagnosis techniques should be designed with the concept of parallelization in mind from the beginning.

Acknowledgements

The authors were supported by the Carinthian Science Fund (KWF) under contract KWF-3520/26767/38701, the Austrian Science Fund (FWF) and the German Research Foundation (DFG) under contract numbers I 2144 N-15 and JA 2095/4-1 (Project "Debugging of Spreadsheet Programs").

References

1. de Kleer, J., Mackworth, A.K., Reiter, R.: Characterizing Diagnoses and Systems. Artificial Intelligence **56**(2-3) (1992) 197–222
2. de Kleer, J., Williams, B.C.: Diagnosing Multiple Faults. Artificial Intelligence **32**(1) (April 1987) 97–130
3. Reiter, R.: A Theory of Diagnosis from First Principles. Artificial Intelligence **32**(1) (1987) 57–95
4. Felfernig, A., Friedrich, G., Jannach, D., Stumptner, M.: Consistency-based Diagnosis of Configuration Knowledge Bases. Artificial Intelligence **152**(2) (2004) 213–234
5. Mateis, C., Stumptner, M., Wieland, D., Wotawa, F.: Model-Based Debugging of Java Programs. In: AADEBUG'00. (2000)
6. Jannach, D., Schmitz, T.: Model-based Diagnosis of Spreadsheet Programs: A Constraint-based Debugging Approach. Automated Software Engineering **23**(1) (2016) 105–144
7. Wotawa, F.: Debugging Hardware Designs Using a Value-Based Model. Applied Intelligence **16**(1) (2001) 71–92
8. Felfernig, A., Friedrich, G., Isak, K., Shchekotykhin, K.M., Teppan, E., Jannach, D.: Automated Debugging of Recommender User Interface Descriptions. Applied Intelligence **31**(1) (2009) 1–14
9. Console, L., Friedrich, G., Dupré, D.T.: Model-Based Diagnosis Meets Error Diagnosis in Logic Programs. In: IJCAI'93. (1993) 1494–1501
10. Friedrich, G., Shchekotykhin, K.M.: A General Diagnosis Method for Ontologies. In: ISWC'05. (2005) 232–246
11. Stumptner, M., Wotawa, F.: Debugging Functional Programs. In: IJCAI'99. (1999) 1074–1079
12. Friedrich, G., Stumptner, M., Wotawa, F.: Model-Based Diagnosis of Hardware Designs. Artificial Intelligence **111**(1-2) (1999) 3–39
13. White, J., Benavides, D., Schmidt, D.C., Trinidad, P., Dougherty, B., Cortés, A.R.: Automated Diagnosis of Feature Model Configurations. Journal of Systems and Software **83**(7) (2010) 1094–1107
14. Friedrich, G., Fugini, M., Mussi, E., Pernici, B., Tagni, G.: Exception Handling for Repair in Service-Based Processes. IEEE Transactions on Software Engineering **36**(2) (2010) 198–215
15. Junker, U.: QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. In: AAAI'04. (2004) 167–172
16. Marques-Silva, J., Janota, M., Belov, A.: Minimal Sets over Monotone Predicates in Boolean Formulae. In: Computer Aided Verification. (2013) 592–607
17. Shchekotykhin, K., Jannach, D., Schmitz, T.: MergeXplain: Fast Computation of Multiple Conflicts for Diagnosis. In: IJCAI'15. (2015) 3221–3228
18. Greiner, R., Smith, B., Wilkerson, R.: A Correction to the Algorithm in Reiter's Theory of Diagnosis. Artificial Intelligence **41**(1) (1989) 79–88
19. Jannach, D., Schmitz, T., Shchekotykhin, K.: Parallel Model-Based Diagnosis On Multi-Core Computers. Journal of Artificial Intelligence Research (JAIR) **55** (2016) 835–887
20. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co. (1979)
21. Eiter, T., Gottlob, G.: The Complexity of Logic-Based Abduction. Journal of the ACM **42**(1) (1995) 3–42
22. de Kleer, J.: Hitting Set Algorithms for Model-based Diagnosis. In: DX'11. (2011) 100–105
23. Stern, R., Kalech, M., Feldman, A., Provan, G.: Exploring the Duality in Conflict-Directed Model-Based Diagnosis. In: AAAI'12. (2012) 828–834
24. Marques-Silva, J., Janota, M., Ignatiev, A., Morgado, A.: Efficient Model Based Diagnosis with Maximum Satisfiability. In: IJCAI'15. (2015) 1966–1972
25. de Kleer, J., Williams, B.C.: Diagnosing Multiple Faults. Artif. Intell. **32**(1) (apr 1987) 97–130
26. Williams, B.C., Ragno, R.J.: Conflict-directed A* and its Role in Model-based Embedded Eystems. Discrete Applied Mathematics **155**(12) (2007) 1562–1595

27. Darwiche, A.: Model-Based Diagnosis using Structured System Descriptions. Journal of Artificial Intelligence Research **8** (1998) 165–222
28. Siddiqi, S., Huang, J.: Sequential Diagnosis by Abstraction. Journal of Artificial Intelligence Research **41** (2011) 329–365
29. Darwiche, A.: A Differential Approach to Inference in Bayesian Networks. Journal of the ACM **50**(3) (May 2003) 280–305
30. Pill, I., Quaritsch, T.: Optimizations for the Boolean Approach to Computing Minimal Hitting Sets. In: ECAI'12. (2012) 648–653
31. Feldman, A., Provan, G., de Kleer, J., Robert, S., van Gemund, A.: Solving Model-Based Diagnosis Problems with Max-SAT Solvers and Vice Versa. In: DX'10. (2010) 185–192
32. Metodi, A., Stern, R., Kalech, M., Codish, M.: A Novel SAT-Based Approach to Model Based Diagnosis. Journal of Artificial Intelligence Research **51** (2014) 377–411
33. Mencia, C., Marques-Silva, J.: Efficient Relaxations of Over-constrained CSPs. In: ICTAI'14. (2014) 725–732
34. Mencía, C., Previti, A., Marques-Silva, J.: Literal-Based MCS Extraction. In: IJCAI'15. (2015) 1973–1979
35. Nica, I., Pill, I., Quaritsch, T., Wotawa, F.: The Route to Success: A Performance Comparison of Diagnosis Algorithms. In: IJCAI'13. (2013) 1039–1045
36. : Interactive ontology debugging: Two query strategies for efficient fault localization. Journal of Web Semantics **12-13** (2012) 88 – 103 Reasoning with context in the Semantic Web.
37. Feldman, A., Provan, G., van Gemund, A.: Approximate Model-Based Diagnosis Using Greedy Stochastic Search. Journal of Artifcial Intelligence Research **38** (2010) 371–413
38. Li, L., Yunfei, J.: Computing Minimal Hitting Sets with Genetic Algorithm. In: DX'02. (2002) 1–4
39. Ram, D.J., Sreenivas, T.H., Subramaniam, K.G.: Parallel Simulated Annealing Algorithms. Journal of Parallel and Distributed Computing **37**(2) (1996) 207 – 212
40. Burns, E., Lemons, S., Ruml, W., Zhou, R.: Best-First Heuristic Search for Multicore Machines. Journal of Artificial Intelligence Research **39** (2010) 689–743
41. Ferguson, C., Korf, R.E.: Distributed Tree Search and its Application to alpha-beta Pruning. In: AAAI'88. (1988) 128–132
42. Brüngger, A., Marzetta, A., Fukuda, K., Nievergelt, J.: The Parallel Search Bench ZRAM and its Applications. Annals of Operations Research **90**(0) (1999) 45–63
43. Kalyanpur, A., Parsia, B., Horridge, M., Sirin, E.: Finding All Justifications of OWL DL Entailments. In: ISWC 2007 + ASWC 2007. (2007) 267–280
44. Previti, A., Ignatiev, A., Morgado, A., Marques-Silva, J.: Prime Compilation of Non-Clausal Formulae. In: IJCAI'15. (2015) 1980–1987
45. Powley, C., Korf, R.E.: Single-agent Parallel Window Search. IEEE Transactions on Pattern Analysis and Machine Intelligence **13**(5) (1991) 466–477
46. Anglano, C., Portinale, L.: Parallel Model-based Diagnosis using PVM. In: EuroPVM'96. (1996) 331–334
47. Wotawa, F.: A Variant of Reiter's Hitting-set Algorithm. Information Processing Letters **79**(1) (2001) 45–51
48. Phillips, M., Likhachev, M., Koenig, S.: PA*SE: Parallel A* for Slow Expansions. In: ICAPS'14. (2014)
49. Korf, R.E., Schultze, P.: Large-scale Parallel Breadth-first Search. In: AAAI'05. (2005) 1380–1385
50. Shchekotykhin, K.M., Friedrich, G., Rodler, P., Fleiss, P.: Sequential Diagnosis of High Cardinality Faults in Knowledge-Bases by Direct Diagnosis Generation. In: ECAI'14. (2014) 813–818
51. Kurtoglu, T., Feldman, A.: Third International Diagnostic Competition (DXC 11). https://sites.google.com/site/dxcompetition2011 (2011) Accessed: 2016-03-15.
52. Prud'homme, C., Fages, J.G., Lorca, X.: Choco Documentation. (2015) http://www.choco-solver.org.
53. Cardoso, N., Abreu, R.: A Distributed Approach to Diagnosis Candidate Generation. In: EPIA'13. (2013) 175–186

54. Abreu, R., van Gemund, A.J.C.: A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis. In: SARA'09. (2009) 2–9
55. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM **51**(1) (2008) 107–113
56. Zhao, X., Ouyang, D.: Deriving All Minimal Hitting Sets Based on Join Relation. IEEE Transactions on Systems, Man, and Cybernetics: Systems **45**(7) (2015) 1063–1076
57. Lin, L., Jiang, Y.: The computation of Hitting Sets: Review and New Algorithms. Information Processing Letters **86**(4) (2003) 177–184